Web Retrieval Programming Homework #1

b00902107 Shu-Hung You

1 Algorithm

I implemented the Vector Space Model (VSM) with Term Frequency–Inverse Document Frequency (tf–idf) weighting. For each query, the title is break up to a list of words to retrieve documents containing any of these words. The retrieved documents are then sorted according to their cosine distance with the keywords in the concepts (which is a field of query). If Racchio Relevant Feedback is turned on, the first few documents is used as relevant documents to adjust the query vector. Currently, the word-breaking algorithm is pretty naive. After first breaking up the input string by the stop words, every consecutive pair of characters is used as a word.

The rationale between this algorithm is that related documents often contains the keyword in the title, yet the title itself does not contain enough information to order the documents. Hence the concepts is used as the query vector to rank the document, while the title acts as a filter removing irrelevant documents.

The tf–idf weighting formula is chosen according to the recommend list on the Wikipedia. In the following equations, $t \in \text{Term}$, $1 \le j \le N$, $n_t = |\{d \in D \mid t \in D\}|$ and $f_{t,j}$ ($f_{t,q}$) denotes the frequency of t in document d_j (the query, respectively).

- Document Weighting: For $\mathbf{d}_j = (w_{1,j}, \ldots, w_{N,j})$,

$$w_{t,j} = f_{t,j} \log \frac{N}{n_t}$$

- Query Term Weighting: For $\mathbf{q} = (w_{1,q}, \ldots, w_{N,q})$,

$$w_{t,q} = \begin{cases} \left(0.5 + 0.5 \dfrac{f_{t,q}}{\max_{t'} f_{t',q}}\right) \log \dfrac{N}{n_t} & f_{t,q} \neq 0 \\ 0 & f_{t,q} = 0 \end{cases}$$

  However, since the (bi-gram) word in the query is rarely repeated, the term weight is effectively just a binary value.

2 Implementation

Preprocessing Since the factor $||q||$ in the cosine distance does not affect ordering, it is omitted in the computation. Also, as $||d_j||$s are independent of the query for all j, they are precomputed and stored in the file `veclen.ss`.

In addition to the inverted index, all documents are also indexed with the words in order to accelerate the computation of Rocchio algorithm. The inverted index is also pre-converted into a format that simplifies the parsing.

Inverted Index Representation The inverted index is transformed into the format that can be read by the Scheme reader. For example, the file `invidx.ss` storing processed inverse indices contains

```
#(#() #(#(33689 38365)) #(#(33256) (12371 . #(33256))) #(#((10849 . 2))
(6756 . #(10849)) (6850 . #(10849))) #(#(33320) (5600 . #(33320))) #(#(32656)
(12374 . #(32656))) #(#(10346) (11029 . #(10346))) #(#(40549) (7510 . #(40549)))
#(#(14607) (5590 . #(14607))) #(#(14993) (12371 . #(14993))) ...
```

where `#( ... )` is the external representation of scheme vectors. To save space, I have also chosen a special representation of the inverted indices. For each file–count pair in the inverted index, only the file ID is stored if the count equals 1. Otherwise, a file–count pair is stored. For example, the file–count sequence `#(1 2 (4 . 5))` indicates that the current word occurs in file 1, 2, 4 with 1, 1 and 5 occurrences, respectively. Since there are only 10011683 out of 36127070 words (bi-gram) that occur more than once, this representation clearly saves a lot.

By using this representation, the reading time of inverted indices shortened from 3 minutes to 25 seconds, and consumes much less memory. Though gap sequences and γ-encoding are not implemented, the current representation is quite enough.

Rocchio Relevance Feedback The feedback formula is first simplified as follows.

$$\widehat{\mathbf{q}} = a\mathbf{q} + b\frac{1}{|D_r|}\sum_{\mathbf{d}\in D_r}\mathbf{d} - c\frac{1}{|D_{nr}|}\sum_{\mathbf{d}\in D_{nr}}\mathbf{d}$$

$$\mathbf{d}_j \cdot \widehat{\mathbf{q}} = a\mathbf{d}_j \cdot \mathbf{q} + b\frac{1}{|D_r|}\sum_{\mathbf{d}_j\in D_r}\mathbf{d}_j \cdot \mathbf{d} - c\frac{1}{|D_{nr}|}\sum_{\mathbf{d}\in D_{nr}}\mathbf{d_j} \cdot \mathbf{d}$$

Thus, it suffices to compute $\mathbf{d}_j \cdot \mathbf{d}$ and adjust the result of the inner product accordingly. Since the documents are previously indexed with the words, we enumerate all the words in $\mathbf{d} \in D_r$ and compute their inner product with $\mathbf{d}_j$.

In my program, a is taken to be 1 and b is taken to be 0.5 while $c = 0$, and the first 3 documents retrieved by the original query are used as relevant documents. However, it takes too long for the current implementation to re-rank all retrieved documents. Hence only the top 20 ranking documents are re-ranked according to the Rocchio algorithm.

3 Evaluation

Several experiments are conducted with the following settings.

1. The documents are retrieved and ranked only by query title. This configuration yields 0.5285 MAP.

2. The documents are retrieved and ranked by concepts. The MAP increases to 0.7078.

3. The documents are retrieved by query title and ranked by concepts. At this time, the concepts are treated as a single string and breaks up using the same algorithm as title. The MAP is 0.7268.

4. The documents are retrieved by query title and ranked by concepts. However, the title and the concepts are first split by the stop words. The resulting MAP increases slightly to 0.7289.

5. The top 20 ranking documents are re-ranked with Rocchio algorithm. The MAP is 0.71.

4 Discussion

From the experiment, it shows that the preciseness and amount of query terms affects the ranking the most, as illustrated between configuration set 1 and 2. The MAP jumps from 0.5 to 0.7 when concepts are involved. However, adding more words in the retrieval process also results in more in irrelevant documents. In the second configuration set, the ranking algorithm almost runs through the whole document set. Since an important document ought to contain the exact keywords, I add the assumption that using merely query title can filter out relevant documents. The result is configuration set 3. Breaking the query strings by stop words slightly increases the performance, this makes a hint that better word-breaking algorithm may give boost the performance.

The Rocchio algorithm, unfortunately, decreases the performance. I did not tune the parameter though, hence the performance might actually be better after tuning.

5 References

1. Tf–idf. `http://en.wikipedia.org/wiki/Tf%E2%80%93idf`
2. Rocchio_algorithm. `http://en.wikipedia.org/wiki/Rocchio_algorithm`